CT421

Artificial Intelligence

Name: Andrew Hayes
Student ID: 21321503
E-mail: a.hayes18@universityofgalway.ie

2025–02–28

Contents

1 Introduction

1.1 Assessment

- Exam: 60%.
- 2 projects: 20% each.

1.2 Introduction to Artificial Intelligence

The field of Artificial Intelligence has evolved & changed many times over the years, with changes focussing both on the problems & approaches in the field; many problems that were considered typical AI problems are now often considered to belong in different fields. There are also many difficulties in defining intelligence: the **Turing test** attempts to give an objective notion of intelligence and abstracts away from any notions of representation, awareness, etc. It attempts to eliminate bias in favour of living beings by focusing solely on content of questions & answers. There are many criticisms of the Turing test:

- Bias towards symbolic problem-solving criticisms;
- Doesn't test many aspects of human intelligence;
- Possibly constrains notions of intelligence.

1.2.1 Approaches to AI

- **Classical AI** uses predicate calculus (& others) and logical inference to infer or find new information. It is a powerful approach for many domains, but issues arise when dealing with noise or contradictory data.
- **Machine learning** learns from data and uses a distributed representation of learned information. It is typified by neural networks & deep learning approaches.
- **Agent-based systems** view intelligence as a collective emergent behaviour from a large number of simple interacting individuals or *agents*. Social systems provide another metaphor for intelligence in that they exhibit global behaviours that enable them to solve problems that would prove impossible for any of the individual members. Properties of agent-based systems / artificial life include:
  - Agents are autonomous or semi-autonomous;
  - Agents are situated;
  - Agents are interactional;
  - Society is structured;
  - Intelligence is emergent.

2 Search

Many problems can be viewed as a **search** problem; consider designing an algorithm to solve a sudoku puzzle: in every step, we are effectively searching for a move (an action) that takes us to a correct legal state. To complete the game, we are iteratively searching for an action that brings us to legal board and so forth until completion. Other examples include searching for a path in a maze, word ladders, chess, & checkers. The problem statement can be formalised as follows:

- The problem can be in various states.
- We start in an initial state.
- There is a set of actions available.
- Each action changes the state.
- Each action has an associated cost.

- We want to reach some goal while minimising cost.

More formally:

- There is a set of (possible/legal) states S;
- There is some start state $s_0 \in S$;
- There is a set of actions A and action rules $a(s) \to s'$;
- There is some goal test $g(s) \to \{0, 1\}$ that tests if we have satisfied our goal;
- There is some cost function $C(s, a, s') \to \mathbb{R}$ that associates a cost with each action;
- Search can be defined by the 5-tuple (S, s, a, g, C).

We can then state the problem as follows: find a sequence of actions $a_1 \dots a_n$ and corresponding states $s_0 \dots s_n$ such that:

- $s_0 = s$
- $s_i = a_i(S_{i-1})$
- $g(s_n) = 1$

while minimising the overall cost $\sum_{i=1}^{n} c(a_i)$.

The problem of solving a sudoku puzzle can be re-stated as:

- Sudoku states: all legal sudoku boards.
- Start state: a particular, partially filled-in, board.
- Actions: inserting a valid number into the board.
- Goal test: all cells filled with no collisions.
- Cost function: 1 per move.

We can conceptualise this search as a **search tree**: a node represents a state, and the edges from a state represent the possible actions from that state, with the edge pointing to the new resulting state from the action. Important factors of a search tree include:

- The breadth of the tree (branching factor).
- The depth of the tree.
- The minimum solution depth.
- The size of the tree $O(b^d)$.
- The **frontier:** the set of unexplored nodes that are reachable from any currently explored node.
- Choosing which node to explore next is the key in search algorithms.

2.1 Uninformed Search

In **uninformed search**, no information is known (or used) about solutions in the tree. Possible approaches include expanding the deepest node (depth-first search) or expanding the closest node (breadth-first search). Properties that must be considered for uninformed search include completeness, optimality, time complexity (the total number of nodes visited), & space complexity (the size of the frontier).

```
visited = {};
frontier = {s_0};
goal_found = False;

while (not goal_found):
    node = frontier.next();
    frontier.delete(node);

    if (g(node)):
        goal_found = True;
    else
        visited.add(node);
        for child in node.children():
            if (not visited.contains(child)):
                frontier.add(child);
```

Listing 1: Pseudocode for an uninformed search

The manner in which we expand the node is key to how the search progresses. The way in which we implement `frontier.next()` determines the type of search; otherwise the basic approach remains unchanged.

Depth-first search is good regarding memory cost, but produces suboptimal solutions:

- Space: $O(bd)$.
- Time: $O(b^d)$.
- Completeness: only for finite trees.
- Optimality: no.

Breadth-first search produces an optimal solution, but is expensive with regards to memory cost:

- Space: $O(b^{m+1})$, where m is the depth of the solution in the tree.
- Time: $O(b^m)$.
- Completeness: yes.
- Optimality: yes (assuming constant costs).

Iterative deepening search attempts to overcome some of the issues of both breadth-first and depth-first search. It works by running depth-first search to a fixed depth of z by starting at $d = 1$ and if no solution is found, incrementing d and re-running.

- Low memory requirements (equal to depth-first search).
- Not many more nodes expanded than breadth-first search.
- Note that the leaf level will have more nodes than the previous layers.

Thus far, we have assumed each edge has a fixed cost; consider the case where the costs are not uniform: neither depth-first search or breadth-first search are guaranteed to find the least-cost path in the case where action costs are not uniform. One approach is to choose the node with the lowest cost: order the nodes in the frontier by cost-so-far (cost of the path from the start state to the current node) and explore the next node with the smallest cost-so-far, which gives an optimal and complete solution (given all positive costs).

2.2 Informed Search

Thus far, we have assumed we know nothing about the search space; what should we do if we know *something* about the search space? We know the cost of getting to the current node: the remaining cost of finding the solution is the cost from the current node to the goal state; therefore, the total cost is the cost of getting from the start state to the current node, plus the cost of getting from the current node to the goal state. We can use a (problem-specific) **heuristic** $h(s)$ to estimate the remaining cost: $h(s) = 0$ is s is a goal. A good heuristic is fast to compute and close to the real costs.

Given that $g(s)$ is the cost of the path so far, the **A* algorithm** expands the node s to minimise $g(s) + h(s)$. The frontier nodes are managed as a priority queue. If h never overestimates the cost, the A* algorithm will find the optimal solution.

2.3 Adversarial Search

The typical game setting is as follows:

- 2 player;
- Alternating turns;
- Zero-sum (gain for one, loss for another);
- Perfect information.

A game is said to be **solved** if an optimal strategy is known.

- A **strong solved** game is one which is solved for all positions;
- A **weak solved** game is one which is solved for some (start) positions.

A game has the following properties:

- A set of possible states;
- A start state;
- A set of actions;
- A set of end states (many);
- An objective function;
- Control over actions alternates.

The **minimax algorithm** computes a value for each node, going backwards from the end-nodes. The **max player** selects actions to maximise return, while the **min player** selects actions to minimise return. The algorithm assumes perfect play from both players. For optimal play, the agent has to evaluate the entire game tree. Issues to consider include:

- Noise / randomness;
- Efficiency – size of the tree;
- Many game trees are too deep;
- Many game trees are too broad.

Alpha-beta pruning is a means to reduce the search space wherein sibling nodes can be pruned based on previously found values. Alpha represents the best maximum value found so far (for the maximising player), and beta represents the best minimum value found so far (for the minimising player). If a node's value proves irrelevant (based on the alpha & beta values), its entire subtree can be discarded. In reality, for many search scenarios in games, even with alpha-beta pruning, the space is much too large to get to all end states. Instead, we use an **evaluation function** which is effectively a heuristic to estimate the value of a state (the probability of a win or a loss). The search is ran to a fixed depth and all states are evaluated at that depth. Look-ahead is performed from the best states to another fixed depth.

Horizon effects refer to the limitations that arise when the algorithm evaluates a position based on a finite search depth (the horizon):

- What if something interesting / unusual / unexpected occurs at horizon + 1?
- How do you identify that?
- When to generate and explore more nodes?
- Deceptive problems.

2.4 Monte Carlo Tree Search

Minimax and alpha-beta pruning are both useful approaches, and alpha-beta pruning effectively results in computing the square root of the branching factor. In many cases, however, game trees have too high a **blocking factor** and evaluating the leaf nodes is not possible; instead, an **estimation function** is used instead.

Monte Carlo tree search continually estimates the value of tree nodes. It combines ideas from classical tree search with ideas from machine learning, and balances the exploration of unseen space with exploitation of the best known move. Monte Carlo tree search works by exploring all potential options at the time. The best move is identified and other option are searched for while validating how good the current action is. It uses **play out**, which is simulation considering random actions.

1. **Selection:** pick a suitable path.
2. **Expansion:** expand from that path.
3. **Simulation:** simulation / play outs.
4. **Back propagation:** update states.

There is a trade-off between exploration and exploitation; the balance is usually controlled by the number of parameters / factors, including the number of wins, simulations, & exploration cost. Monte Carlo grid search is used in a large number of domains, including AlphaGo, DeepMind, & many video games.

2.5 Local Search Algorithms

In many domains, we are only interested in the goal state: the path to finding the goal is irrelevant. The main idea in **local search algorithms** is to maintain the current state and repeatedly try to improve upon the current state. This results in little memory overhead and can find reasonable solutions even in large or infinite spaces. Local search algorithms are suitable only for certain classes of problems in which maximising (or minimising) certain criteria among a number of candidate solutions is desirable. Local search algorithms are **incomplete algorithms**, as they may not find the optimal solution.

2.6 Hill Climbing

Hill climbing involves moving in the direction of increasing value and stopping when a peak is reached. There is no look-ahead involved, and only the current state & the objective function are maintained. One problem with hill climbing algorithms is local maxima & minima; therefore, the success of the algorithm depends on the shape of the search space. One approach to solve this problem is **random restart**.

```
current_node = initial_state;

while (stopping_criteria):
    neighbour = current_node.fittest_neighbour;

    if neighbour.value > current_node.value:
        current_node = neighbour;
```

Listing 2: Hill climbing pseudocode

One problem that can be solved with a hill climbing algorithm is the **8 queens problem**: place 8 queens on a chess board so that no two queens are attacking each other. The problem can be stated more formally as: given an 8×8 grid, identify 8 squares so that no two squares are on the same row, column, or diagonal. The problem can also be generalised to an $N \times N$ board. An algorithm can be generated using hill climbing to find a solution. However, it must be noted that the algorithm may get caught at a local maximum.

Simulated annealing attempts to avoid getting stuck in local maxima by randomly moving to another state. The move is based on the fitness of the current state, the new state, and the **temperature** (a parameter which decreases over time and reduces the probability of changing states).

2.7 Well-Known Optimisation Problems

2.7.1 Knapsack Problem

There are a set of items, each with a value & a weight. The goal is to select a subset of items such that the weight of these items is below a threshold and the sum of the values is optimised / maximised. The problem is how to select those items. More formally, given two n-tuples of values $< v_0, v_1, \ldots, v_n >$ and $< w_0, w_1, \ldots, w_n >$, choose a set of items i from the $n$items such that $\sum_i v(i)$ is maximised and $\sum_i w(i) \leq T$.

The brute force approach is to enumerate all subsets and keep the subset that gives the greatest payoff: $O(2^n)$. The greedy approach is more efficient but won't give the best solution.

There are many variations on the knapsack problem; the variation described above is the 0/1 knapsack problem; there are other scenarios where part of an item may be taken, variations wherein there are constraints over the items where the value of an item is dependent on another item being chosen or not, and variations with multiple knapsacks.

2.8 Travelling Salesman Problem

Given N cities, devise a tour that involves visiting every city once. The distance, or *cost*, between pairs of cities is known and the goal is to minimise the cost of the tour. There are many applications of this problem in scheduling & optimisation.

The brute force approach is to enumerate all tours and choose the cheapest, and is computationally intractable.

3 Genetic Algorithms

Genetic algorithms are directed search algorithms inspired by biological evolution, developed by John Holland in the 1970s to study the adaptive processes occurring in natural systems. They have been used as search mechanisms to find good solutions for a range of problems. At a high level, the algorithm is as follows:

1. Produce an initial population of individuals.
2. Evaluate the fitness of all individuals.
3. While the termination condition is not met, do:

1. Select the fitter individuals for reproduction.
2. Recombine between individuals.
3. Mutate individuals.
4. Evaluate the fitness of the modified individuals.
5. Generate a new population.

There are many potential options for the representation of chromosomes, including bit strings, integers, real numbers, lists of rules/instructions, & programs. To initialise the population, we typically start with a population of randomly generated individuals, but we can also use a previously-saved population, a set of solutions provided by a human expert, or a set of solutions provided by another algorithm. As rules of thumb, we generally use a data structure as close as possible to the natural representation, write appropriate genetic operators as needed, and, if possible, ensure that all genotypes correspond to feasible solutions. Selection can be fitness-based (roulette wheel), rank-based, tournament selection, or elitism.

Crossover is a vital component of genetic algorithms that allows the recombination of good sub-solutions and speeds up search early in evolution. **Mutation** represents a change in the gene; the main purpose is to prevent the search algorithm from becoming trapped in a local maxima.

As a simple example, suppose that one wishes to maximise the number of `1`s in a string of l binary digits. Similarly, we could set the target to be an arbitrary string of `1`s and `0`s. An individual can be represented as a string of binary digits. The fitness of a candidate solution is the number of `1`s in its genetic code. A related problem, maintaining the same representation as before, is to modify the fitness function as follows: for strings of length l with x `1`s, if $x > 1$, then fitness is equal to x; else, fitness is equal to $l + k$ for $k > 0$.

3.1 Schema Theorem

Consider a genetic algorithm using only a binary alphabet. $\{0, 1, *\}$ is the alphabet, where $*$ is a special wildcard symbol which can be either 0 or 1. A **schema** is any template comprising a string of these three symbols; for example the schema $[1 * 1*]$ represents the following four strings: $[1010]$, $[1011]$, $[1110]$, $[1111]$.

The **order** of a schema S is the number of fixed positions (0 or 1) presented in the schema. For example, $[01 * 1*]$ has order 3, $[1 * 1 * 10 * 0]$ has order 5. The order of a schema is useful in estimating the probability of survival of a schema following a mutation.

The **defining length** of a schema S is the distance between the first and last fixed positions in the schema. For example, the schema $S_1 = [01 * 1*]$ has defining length 3. The defining length of a schema indicates the survival probability of the schema under crossover.

Given selection based on fitness, the expected number of individuals belonging to a schema S at generation $i + 1$ is equal to the number of them present at generation i multiplied by their fitness over the average fitness. An "above average" schema receives an exponentially increasing number of strings over the evolution. The probability of a schema S surviving crossover is dependent on the defining length: schemata with above-average fitness with short defining lengths will still be sampled at exponentially increasing rates. The probability of a schema S surviving mutation is dependent on the order of the schema: schemata with above-average fitness with low orders will still be sampled at exponentially increasing rates.

The **schema theorem** states that short, low-order, above-average schemata receive exponentially increasing representation in subsequent generations of a genetic algorithm. The **building-block hypothesis** states that a genetic algorithm navigates the search space through the re-arranging of short, low-order, high-performance schemata, termed *building blocks*.

3.2 Landscapes

A **landscape** is a visualisation of the relationship between genotype & fitness; it can give an insight into the complexity of the problem at hand. Landscapes can be adaptive.

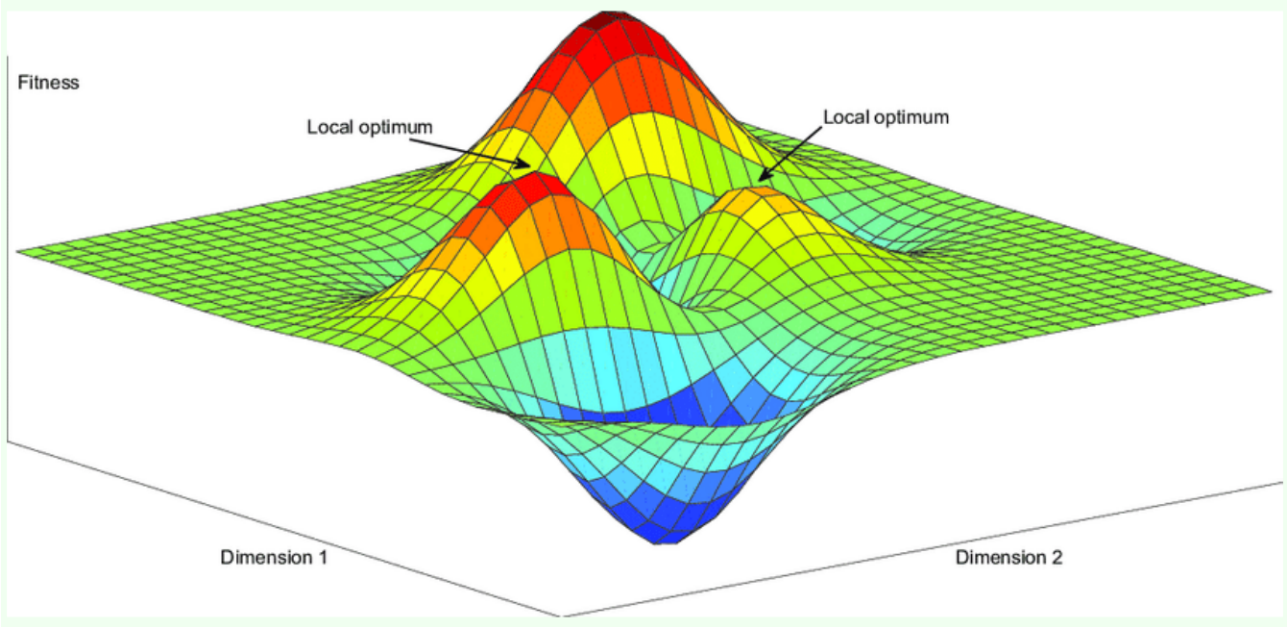

Figure 1: Fitness landscape example. The peaks on the landscape represent high fitness and hence the ability of the genotype to survive. The valleys or troughs indicate low fitness.

An **NK fitness landscape** is a model of genetic interactions, developed to explain & explore the effects of local features on the ruggedness of a fitness landscape – *ruggedness* plays a key roles in ascertaining how difficult it is to find the global optimum. NK landscapes allow us to tune the ruggedness. Each component (gene) of the solution space makes a contribution to the fitness; the contribution to the landscape depends on the value of that gene itself but also on the state of K other nodes, where K can be changed to give different landscapes. If $K = 0$, all genes are independent and this is typically a smooth multi-modal landscape; as K increases, the landscape becomes more rugged.

One approach to create NK fitness landscapes is to use a *lookup table* of size 2^K where each row in the lookup table represents the neighbourhood values and the fitness achieved. Variations on NK fitness landscapes can be made by using non-uniform interaction sizes or allowing non-adjacent genes to influence each other's fitness.

Fitness clouds can be created by randomly sampling the population, generating K mutated versions of the sampled genotypes, measuring their fitness, and plotting their fitness over time, thus giving insight into the landscape.

3.3 Objective/Fitness Functions

We usually specify the objective in the fitness function, for example, the thing we are trying to maximise or minimise or some constraint that we want to satisfy. This can be very difficult, and sometimes we don't even know how to specify the function; furthermore, fitness functions can be costly to evaluate. Issues arise with this:

- "Most ambitious objectives don't illuminate a path to themselves."
- "Many great discoveries are not the result of objective-driven search."
- "Natural evolution innovates through an open-ended process that lacks a final objective."
- "Searching for a fixed objective, the dominant paradigm in EC and ML, may ultimately limit what can be achieved."

The more ambitious the objective fitness function, the less likely it is that evolution will solve it. The two big issues with fitness landscapes (neutral plains and ruggedness) can both be attributed, at least in part, to the fitness function

3.4 Diversity

It's important to maintain diversity in the population for genetic algorithms. Once a population converges on a local optima, it can be difficult to introduce sufficient diversity to climb out of local optima. Many approaches have been proposed to maintain diversity. If diversity decreases, then a big increase in mutation levels called **hypermutation** can be used in the hopes of introducing novelty. Then, we need some measure of diversity: it can be measured at the genotypic, phenotypic, or fitness levels.

Co-evolution is often used as a means to help diversity where interactions between individuals contribute to the fitness with the goal that a form of competition will lead to better performance. Alternative representations can also be used to encourage greater diversity by building redundancy into the representation:

- **Multi-layered GA:** add an extra layer or layers between the genotype and the phenotype, thus allowing multiple genotypes to map to a phenotype. This can allow multiple mutations to occur which aren't immediately represented in the phenotype, maintaining increased diversity.
- **Diploid representations:** represent each chromosome by two genetic sequences, one of which is subject to evolutionary pressures, the pother following a random walk. Periodically, a small percentage of chromosomes swap their sequences.
- **Island models for the GA:** partition the population of solutions into sub-groups, with each sub-group evolving separately. Periodically, some solutions are swapped among the separate populations.

Several approaches have been attempted to make the rates of mutation and crossover subject to evolution itself: **self-adaptation**. For example, add a gene to each chromosome which represents the rate at which mutation should be applied to that chromosome or solution. The goal is that the evolutionary process itself will find a suitable mutation rate.

3.5 Novelty Search

The central thesis of **novelty search** is that by solely evolving according to an objective function, we decrease creativity, novelty, & innovation. It argues that this is because many objective functions are deceptive and that we should instead reward solutions (or sub-solutions) that are unique and phenotypically novel. It has been successfully applied in a range of domains including the evolution of movement for robots. In many domains, novelty search has out-performed searching directly for an objective. The standard approach to novelty search involves maintaining an archive of previously-found novel solutions. To decide are the size of the archive, the similarity measure, and the balance between novelty & fitness.

4 Game Theory

4.1 Reasoning about Interactions

Assume that we have just two agents, i and j, and that these agents are self-interested. Let there be a set of "outcomes" $\Omega = \{\Omega_1, \Omega_2, \ldots, \Omega_n\}$ over which the agents have preferences. Preferences are expressed by utility functions:

$$u_i : \Omega \to \mathbb{R}$$
$$u_j : \Omega \to \mathbb{R}$$

These functions lead naturally to preference orderings over outcomes:

$$\Omega \geq u_i \Omega' \to u_i(\Omega) \geq u_i(\Omega')$$

We need a model of the environment in which agents can act. Let us assume agents act simultaneously to choose an action to perform, and as a result of the actions an outcome will result. The actual outcome depends on the combination of actions. This can be represented as a **state transformation function**:

$$\tau : \text{Action}_i \times \text{Action}_j \to \Omega$$

For the time being, we will make the simplifying assumption that an agent can make one of two actions: to co-operate C or to defect D. We say a certain move is **rational** if the outcomes that arise through the action are better than all outcomes that arise from the alternative action.

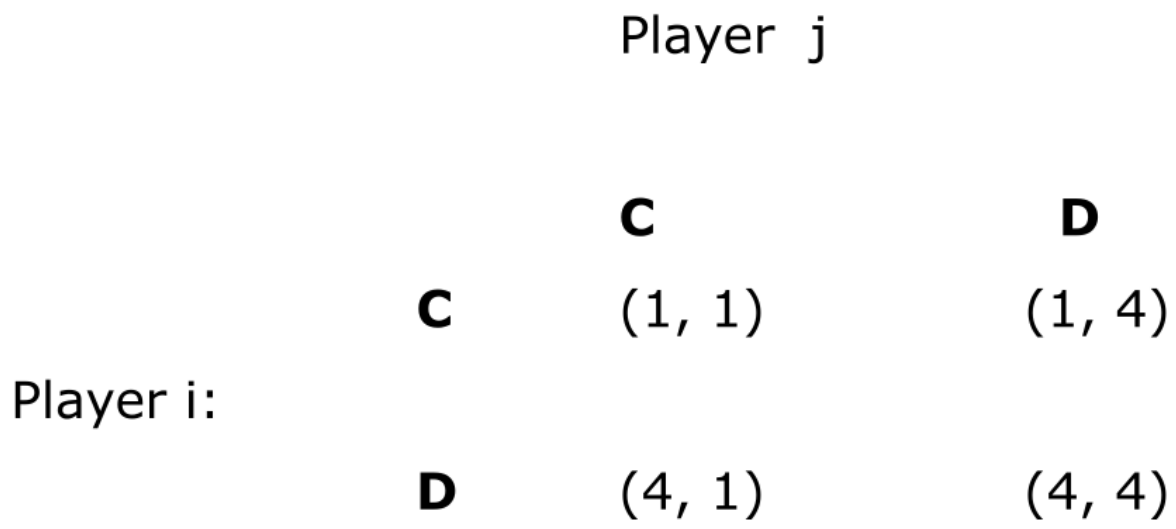

Figure 2: For player j, D is the rational choice

4.2 Dominant Strategy

Given a particular strategy s for agent i, there will be a number of possible outcomes. We say s_1 dominates s_2 if every outcome possible by agent i playing s_1 is preferred over every possible outcome by agent i playing s_2. A rational agent will never play a dominated strategy. However, there is not usually a unique undominated strategy.

4.3 Nash Equilibrium

Two strategies s_1 and s_2 are in **Nash equilibrium** if:

- Assuming agent i plays s_1, agent j can do no better than play s_2; and
- Assuming agent j plays s_2, agent i can do no better than play s_1.

In Nash equilibrium, neither agent has any incentive to deviate from their strategy. Not all possible interactions have a Nash equilibrium, and some interactoins can have several Nash equilibria.

4.4 Prisoner's Dilemma

The **Prisoner's Dilemma** is usually expressed in terms of pay-offs (or rewards) for co-operating or defecting:

		Player j	
		C	D
Player i	C	$(3, 3)$	$(0, 5)$
	D	$(5, 0)$	$(1, 1)$

- If both co-operate, they each get a reward of 3.
- If both defect, they each get a reward of 1.
- If one co-operates and the other defects, the com-operators gets 0 (the sucker's payoff) and the other gets 5.

The individually rational action is to defect: it guarantees a payoff of no worse than 1, whereas co-operating guarantees a payoff of no worse than 0. So, defection is the best response to all strategies; however, common sense indicates that this is not the best response.

The prisoner's dilemma occurs in many domains and is suitable for modelling large classes of multi-agent interactions. There have been many real-world scenarios that are implicitly prisoner's dilemmas (or variations):

- Arms race;
- Environmental issues;
- Free-rider systems;
- Warfare;
- Behaviour in many biological systems — bats, guppie fish, etc;
- Competition between nodes in a distributed computer system;
- Modelling competition and collaboration between information providers;
- Sports.

Variations on the prisoner's dilemma include:

- N**-player dilemma:** for example, the voter's paradox, where it is true that a particular endeavour would return a benefit to all members where each individual would receive rewards; it is also true that any member would receive an even greater reward by contributing nothing. Elections, environment actions, and the tragedy of the commons are all examples of this phenomenon.
- **Spatial organisations:** where agents are placed in some 2-dimensional space and can only interact with neighbours.
- **Partial co-operation:** acts are no longer co-operative or non-co-operative, but can be in some range. If we consider extending the classical IPD to this domain, we can define landscapes using pay-off equations.
- **Noise:** problems arise if we introduce any degree of noise, which will lead co-operations to be interpreted as defections, etc. Consider two TFTs playing witha degree of noise.

Summay so far:

- We need a means to organise & co-ordinate agents. There are underlying problems here with respect to co-operation.
- Game theory & extensions provides a tool to reason about and to develop multi-agent systems.
- We assume agents have a rational ordering of possible outcomes and a set of actions they may choose to bring about those outcomes.
- We have limited the types of interactions to very simple cases.

One extension is the **ultimatum game**. We are no longer just discussing outcomes for simple choices:

- Two players i and j.
- The goal is to distribute some resource, e.g., €100.
- Player i picks a number x, in a range (0-100).
- Player j must accept or reject the offer.
- If Player j rejects: both get 0.

- If Player j accepts: Player i gets x and Player j gets $100 - x$.

This allows us to reason about more complex scenarios. Many extensions are available and have been researched. If we wish to reason about two or more agents/systems agreeing on value for some exchange (information, service), we can look to auction theory. To reason about more complex scenarios, negotiation & argumentation theory has been adopted.

4.5 Auction Theory

Auction theory can be used as a method to allow agents to arrive at an agreement regarding events & actions when agents are self-interested. In some cases, no agreement is possible at all. However, in most scenarios, there is the potential to arrive at a mutually beneficial agreement. There are several approaches that have been adopted to do this; all can bee seen as a form of negotiation or argumentation by the agents. Negotiation or argumentation is governed by some protocol or mechanism: this protocol defines how the agents are to interact, i.e., the actual rules of encounter. Questions that arise include:

- How to design a protocol such that certain properties exist?
- How to design strategies for agents to use a given set of protocols?

Desired features from protocols include: guaranteed success, simplicity, maximising social utility, pareto-efficiency, & individual rationality. **Auctions** represent a class of useful protocols, and are used in many domains. An auction takes place between an agent (auctioneer) and a set of other agents (bidders). The goal is to allocate the goods to one of the bidders. Usually, an auctioneer attempt to maximise the price; the bidders desire to minimise the price. We can categorise auctions according to a range of features:

- Bids may be:
  - Open-cry;
  - Sealed bid.
- Bidding may be:
  - One shot;
  - Ascending;
  - Descending.

Selling goods by auction is more flexible than setting a fixed price and less time-consuming than explicit negotiation (haggling). In many domains, the value of an item may vary enough to preclude direct & absolute pricing. It is a pure form of market; it is efficient in that auctions usually ensure goods are allocated to those who value them most. The price is set, not by the sellers, but by the buyers. No one auction protocol is the best; some are preferred by sellers, others by buyers. Some auctions attempt to prevent cheating, or at least decrease the incentive to cheat; others provide several means to cheat. People tend to bid in auctions for two reasons:

- They wish to acquire the goods (bases bid on private evaluation).
- They wish to acquire the goods to re-sell (bases bid on private evaluation and estimates on future valuations).

4.5.1 English Auction

In an **English auction**, the auctioneer begins with the lowest acceptable price (reserve), and proceeds to obtain successively higher bids from bidders until no-one will increase the bid. It is effectively first-price, open-cry, & ascending. The dominant strategy is to successively bid a small amount more than the current highest id until it reaches their valuation, then withdraw. Potential problems with English auctions include:

- Rings;
- Shills in the bidders;
- Winner's curse.

In some English auctions, the reserve price is kept secret to attempt to prevent rings from forming.

4.5.2 Dutch Auction

In a **Dutch auction**, bidding starts at an artificially high price. Lower prices are offered, in descending order, until a bidder equals to the current price. Goods are then sold to the bidder for that price Dutch auctions are descending, open-cry auctions. From a seller's perspective, the key to a successful auction is the effect of competition on the bidders. In an English auction, a winner may pay well under their valuation and thus the seller loses out; this is not the case in a Dutch auction.

4.5.3 First-Price, Sealed Bid

First-price, sealed bid auctions are usually one-shot auctions. Each bidders submits a sealed bid. The goods are sold to the highest bidders. Best strategy is to bid to true valuation. Interesting variations exist if there are a number of goods to be sold and a number of rounds.

4.5.4 Vickrey Auction

A **Vickrey auction** is a sealed-bid, second-price auction. The price paid by the winner is that price offered by the second-placed bidder. In this type of auction, contrary to initial intuition, sellers make as much, if not more than the first-price auctoins. In reality, bidders are not afraid to bid high, knowing that they will have to pay the second price; bidders tend to be more competitive.

Other auction types exist also: reverse auctions, double auctions, haphazard (whisper auction, handshake auction), etc. We can use auctions as a means to allow agents to agree on a price for buying goods or services. Depending on the type of auction chosen, we will favour buyers or sellers. We sill have some problems though:

- Are auctions the best way?
- What happens following an auction, if upon receiving goods, one doesn't pay?
- What happens following an auction, if upon paying, one realise that the goods are not as expected?
- Is it possible to prevent shills, rings, & other forms of manipulation?
- In auctions, agents agree on a price; can we deal with more dimensions of negotiatoin?